COSC 30203, Fall 2024
Data Lab: Manipulating Bits
Assigned: September 10, Due: September 23, 11:59 PM

1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of integers and floating point numbers. You'll do this by solving a series of programming "puzzles." Many of these puzzles are quite artificial, but you'll find yourself thinking much more about bits in working your way through them.

2 Logistics

This is an individual project. All handins are electronic. Clarifications and corrections will be sent to the class via TCU email.

3 Handout Instructions

The project repository is distributed via GitHub Classroom. If you are reading this, then you have successfully downloaded your project.

The only file you will be modifying and turning in is `bits.c`.

The `bits.c` file contains a skeleton for each of the 22 programming puzzles. Your assignment is to complete each function skeleton using only *straightline* code for the integer puzzles (i.e., no loops or conditionals) and a limited number of C arithmetic and logical operators. Specifically, you are *only* allowed to use the following eight operators:

```
 ! ~ & ^ | + << >>
```

A few of the functions further restrict this list. Also, you are not allowed to use any constants longer than 8 bits. If you need a constant longer than 8 bits, then you must build it using the operators listed above. See (read) the comments in `bits.c` for detailed rules and a discussion of the desired coding style.

4 The Puzzles

This section describes the puzzles that you will be solving in `bits.c`.

Table 1 lists the puzzles in order of difficulty from easiest to hardest. The "Rating" field gives the difficulty rating (the number of points) for the puzzle, and the "Max ops" field gives the maximum number of operators you are allowed to use to implement each function. See the comments in `bits.c` for more details on the desired behavior of the functions. You may also refer to the test functions in `tests.c`. These are used as reference functions to express the correct behavior of your functions, although they don't satisfy the coding rules for your functions.

Name	Description	Rating	Max ops
`bitOr(x,y)`	`x | y` using only `&` and `~`.	1	8
`evenBits()`	return word with all even-numbered bits set to 1	1	8
`fitsShort(x)`	return 1 if x fits in 16 bits	1	8
`thirdBits()`	return word with every third bit set to 1	1	8
`tmin()`	return min 2's complement integer	1	4
`allOddBits(x)`	True only if all odd-numbered bits in `x` set to 1.	2	12
`copyLSB(x)`	set all bits of result to least significant bit of x	2	5
`floatAbsVal(uf)`	return absolute value of floating point argument uf	2	10
`isPositive(x)`	return 1 if x ¿ 0, return 0 otherwise	2	8
`leastBitPos(x)`	return a mask of position of least significant 1 bit	2	6
`oddBits()`	return word with all odd-numbered bits set to 1	2	8
`conditional(x, y, z)`	same as x ? y : z	3	16
`isGreater(x, y)`	if x $>$ y return 1 else 0	3	24
`multFiveEighths(x)`	multiply by 5/8 round to 0	3	12
`remainderPower2(x,n)`	compute $x\%(2^n)$	3	20
`subtractionOK(x,y)`	determine if can compute x-y without overflow	3	20
`bitCount(x)`	return number of 1's in word	4	40
`bitReverse(x)`	reverse bits in 32-bit word	4	45
`greatestBitPos(x)`	return a mask of position of most significant 1 bit	4	70
`intLog2(x)`	return $\lfloor \lg x \rfloor$	4	90
`leftBitCount(x)`	return number of consecutive 1's from left side of word	4	50
`twosCompSignMag(x)`	convert from two's complement to sign-magnitude	4	15

Table 1: Datalab puzzles. For the floating point puzzles, value `f` is the floating-point number having the same bit representation as the unsigned integer `uf`.

For the floating-point puzzles, you will implement some common single-precision floating-point operations. For these puzzles, you are allowed to use standard control structures (conditionals, loops), and you may use both `int` and `unsigned` data types, including arbitrary unsigned and integer constants. You may not use any unions, structs, or arrays. Most significantly, you may not use any floating point data types, operations, or constants. Instead, any floating-point operand will be passed to the function as having type `unsigned`, and any returned floating-point value will be of type `unsigned`. Your code should perform the bit manipulations that implement the specified floating point operations.

The included program `fshow` helps you understand the structure of floating point numbers. To compile `fshow`, switch to the handout directory and type:

```
unix> make
```

You can use `fshow` to see what an arbitrary pattern represents as a floating-point number:

```
unix> ./fshow 2080374784

Floating point value 2.658455992e+36
Bit Representation 0x7c000000, sign = 0, exponent = f8, fraction = 000000
Normalized.  1.0000000000 X 2^(121)
```

You can also give `fshow` hexadecimal and floating point values, and it will decipher their bit structure.

5 Evaluation

Your score will be computed out of a maximum of 100 points based on the following distribution:

56 Correctness points.

44 Performance points.

Correctness points. The puzzles you must solve have been given a difficulty rating between 1 and 4, such that their weighted sum totals to 56. We will evaluate your functions using the `btest` program, which is described in the next section. You will get full credit for a puzzle if it passes all of the tests performed by `btest`, and no credit otherwise.

Performance points. The main concern at this point in the course is that you can get the right answer. However, you should have a sense of keeping things as short and simple as you can. Furthermore, some of the puzzles can be solved by brute force, but this is not advised and you shold be more clever. Thus, for each function there is an established maximum number of operators that you are allowed to use for each function. This limit is very generous and is designed only to catch egregiously inefficient solutions. You will receive two points for each correct function that satisfies the operator limit.

Autograding your work

There are included autograding tools in the datalab directory — `btest`, `dlc`, and `driver.pl` — to help you check the correctness of your work.

- **`btest`:** This program checks the functional correctness of the functions in `bits.c`. To build and use it, type the following two commands:

```
unix> make
unix> ./btest
```

Notice that you must rebuild `btest` each time you modify your `bits.c` file.

You'll find it helpful to work through the functions one at a time, testing each one as you go. You can use the `-f` flag to instruct `btest` to test only a single function:

```
unix> ./btest -f bitXor
```

You can feed it specific function arguments using the option flags `-1`, `-2`, and `-3`:

```
unix> ./btest -f bitXor -1 4 -2 5
```

Check the file `README` for documentation on running the btest program.

- **`dlc`:** This is a modified version of an ANSI C compiler from the MIT CILK group that you can use to check for compliance with the coding rules for each puzzle. The typical usage is:

  ```
  unix> ./dlc bits.c
  ```

  The program runs silently unless it detects a problem, such as an illegal operator, too many operators, or non-straightline code in the integer puzzles. Running with the `-e` switch:

  ```
  unix> ./dlc -e bits.c
  ```

  causes `dlc` to print counts of the number of operators used by each function. Type `./dlc -help` for a list of command line options.

- **`driver.pl`:** This is a driver program that uses `btest` and `dlc` to compute the correctness and performance points for your solution. It takes no arguments:

  ```
  unix> ./driver.pl
  ```

  Your final project submission will use `driver.pl` to evaluate your solution.

6 Handin Instructions

All handins will be through GitHub Classroom. Be sure your functions are well documented. Make sure your name is in a comment box at te type of the file `bits.c`.

```
unix> git status
unix> git add .
unix> git commit -m "put meaningful comment here"
unix> git push
```

7 Advice

- Don't include the `<stdio.h>` header file in your `bits.c` file, as it confuses `dlc` and results in some non-intuitive error messages. You will still be able to use `printf` in your `bits.c` file for debugging without including the `<stdio.h>` header, although `gcc` will print a warning that you can ignore.
- The `dlc` program enforces a stricter form of C declarations than is the case for C/C++ or that is enforced by `gcc`. In particular, any declaration must appear in a block (what you enclose in curly braces) before any statement that is not a declaration. For example, it will complain about the following code:

```
int foo(int x)
{
  int a = x;
  a *= 3;     /* Statement that is not a declaration */
  int b = a;  /* ERROR: Declaration not allowed here */
}
```

8 The "Beat the Prof" Contest

For fun, there is an optional "Beat the Prof" contest that allows you to compete with other students and the instructor to develop the most efficient puzzles. The goal is to solve each Data Lab puzzle using the fewest number of operators. Students who match or beat the instructor's operator count for each puzzle are winners!

To submit your entry to the contest, type:

```
unix> ./driver.pl -u "Your Nickname"
```

Nicknames are limited to 35 characters and can contain alphanumerics, apostrophes, commas, periods, dashes, underscores, and ampersands. You can submit as often as you like. Your most recent submission will appear on a real-time scoreboard, identified only by your nickname. You can view the scoreboard by pointing your browser at

```
http://babbage.cs.tcu.edu:30203
```